263-2300-00: How To Write Fast Numerical Code
Assignment 3: 100 points
Due Date: Th, March 28th, 17:00
https://acl.inf.ethz.ch/teaching/fastcode/2019/
Questions: fastcode@lists.inf.ethz.ch

Submission instructions (read carefully):

- (Submission)
  Homework is submitted through the Moodle system https://moodle-app2.let.ethz.ch/course/view.php?id=10968.
- (Late policy)
  You have 3 late days, but can use at most 2 on one homework, meaning submit latest 48 hours after the due time. For example, submitting 1 hour late costs 1 late day. Note that each homework will be available for submission on the Moodle system 2 days after the deadline. However, if the accumulated time of the previous homework submissions exceeds 3 days, the homework will not count.
- (Formats)
  If you use programs (such as MS-Word or Latex) to create your assignment, convert it to PDF and name it homework.pdf. When submitting more than one file, make sure you create a zip archive that contains all related files, and does not exceed 10 MB. Handwritten parts can be scanned and included or brought (in time) to Alen's, Tyler's or Gagandeep's office.
- (Plots)
  For plots/benchmarks, **provide (concise) necessary information for the experimental setup (e.g., compiler and flags) and always briefly discuss the plot and draw conclusions**. Follow (at least to a reasonable extent) the small guide to making plots from the lecture.
- (Code)
  When compiling the final code, ensure that you use optimization flags (e.g. for GCC use the flag "-O3").
- (Neatness)
  5% of the points in a homework are given for neatness.

Exercises:

1. *Matrix-matrix multiplication kernel (50 pts)*

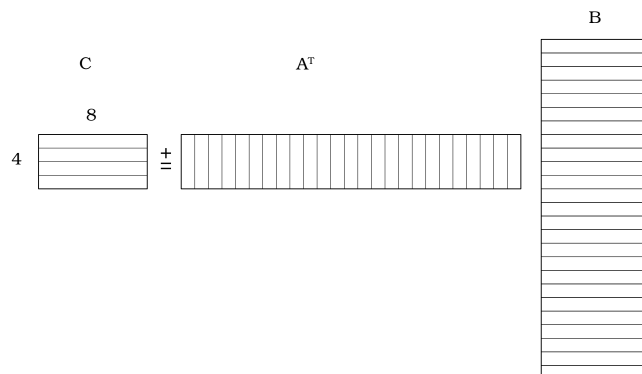
   Consider the following matrix-multiplication operation: $C := A^T B$, where C is a 4×8 row-major matrix and B is an $n \times 8$ row-major matrix. The matrix A^T can be equivalently described as the transpose of an $n \times 4$ row-major matrix A, or as a $4 \times n$ column-major matrix.

   All matrices are double precision.

   This operation used as a so-called *microkernel* in many high-performance linear algebra libraries.

   A skeleton and sample C code that implements this operation is provided here.

   The operation is illustrated below:



Implement the specified matrix-matrix multiplication operation with vector intrinsics using the **AVX2** (with FMA) instruction set. Optimize it as much as you can. You may may use shuffle intrinsics such as `_m256_shuffle_pd` and `_m256_permute2f128_pd`. **For this assignment you are not allowed to use broadcast intrinsics like `_m256_broadcast_pd`.**

Hints:

(a) The size of C is fixed to 4×8, but n, the width of A^T and B, can be any size.

(b) Each iteration of the loop exposes a column of A^T and a row of B, and performs an *outer-product*, where each element of the column of A^T is multiplied with each element of the row of B, updating the corresponding element of C. If a_i is the element in the ith row of the current column of A^T, and b_j is the element in the jth column of the current row of B, then the product $a_i \cdot b_j$ is added to the element in the ith row and j^{th} column of C.

(c) You may find that it is easiest to vectorize this outer-product rather than the loop around it. This way n does not need to be divisible by 4.

Answer the following:

(a) Report the number of flops per cycle attained by your code in a plot for $n = 25, \ldots, 500$ in steps of 25.

(b) What percentage of peak Gflop/cycle does your code attain? Consider for the peak only the adds and mults being performed.

(c) Submit your optimized `microkernel.cpp` file to moodle.

2. *Complex representation conversion (45 pts)*

You are given an array of n **nonzero** complex numbers, stored in the following *interleaved* format:

$$[a_0, b_0, a_1, b_1, a_2, b_2, ..., a_{n-1}, b_{n-1}],$$

where each pair (a_j, b_j) represents the complex number $a_j + b_j i$, $i = \sqrt{-1}$. Your task is to compute an array containing a different (polar-like) representation of these complex numbers, stored in the following interleaved format:

$$[p_0, q_0, p_1, q_1, p_2, q_2, ..., p_{n-1}, q_{n-1}],$$

where $p_j = \frac{a}{b}$, and $q_j = \text{sgn}(a) \cdot \left(a_j^2 + b_j^2\right)$. We define the sign function $\text{sgn}(x)$ as follows:

$$\text{sgn}(x) = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ 1 & x > 0 \end{cases}$$

A skeleton and sample C code that implements this operation is provided here.

(a) Implement the specified operation with vector intrinsics using the **AVX2** (with FMA) instruction set to run as fast as possible.

(b) Report the number of flops per cycle attained by your code.

(c) Considering only the port and throughput information for the SIMD instructions in your conversion routine, give a hard lower bound of its runtime on Haswell.

(d) Submit your `complex_conversion.cpp` file to moodle.